# JPMorgan daily trade Report Engine

<https://www.linkedin.com/in/ignaciopaz/>

## Goal

In terms of programming, I assume that the goal of the exercise is to create an algorithm with good time complexity that can scale as input grows. Also to provide a design that is maintainable, testable and readable.

## Run

**Program**: Just Run java class **jpmorgan.dailytrade.Main**

Simple Output:

|  |
| --- |
| #### Daily incoming/outgoing report ####  04 Jan 2016 - Incoming: $0.00 - Outgoing: $10,025.00  07 Jan 2016 - Incoming: $14,899.50 - Outgoing: $0.00  09 Sep 2018 - Incoming: $0.00 - Outgoing: $10,025.00  30 Sep 2018 - Incoming: $6,782.50 - Outgoing: $1.50  08 Oct 2018 - Incoming: $5,012.50 - Outgoing: $10,025.00  #### Ranking of entities based on incoming ####  bar $21,682.00  foo $5,012.50  bar1 $0.00  foo2 $0.00  #### Ranking of entities based on Outgoing ####  foo $15,037.50  bar1 $10,025.00  foo2 $5,012.50  bar $1.50 |

**Unit tests**: run all tests with maven.

## Questions and Assumptions

These are some questions I would discuss with my own assumptions as answers:

1. **Instructions format**: What is the format of the instructions? It is a file, a service where customer send the data? I assume my program will just receive a collection of instructions
2. **Instructions Batch**: Are the instructions coming altogether in a batch or individual instructions at different date and times? For simplicity I assume it is a one-time batch of instructions.
3. **Instructions size**: How many instructions will be received and for what time period? Thousands of instructions, should be fine to manage in memory. Time frame could be decades.
4. **Raw data sorted**: Are the instructions data sorted? Not sorted at all.
5. **Instruction Date**: It is used for this report? Not at all. The accountability and reports are based on a valid settlement date.
6. **Fields validations**: Regarding the fields of the instructions, can some of them be null, empty or negative? All fields must have a value. Numeric values must be greater or equal than zero. Units must be greater than zero. Instruction date is not used so do not need to validate.
7. **Amounts:** How big are the amounts and totals? Not very big, a double should handle it faster. No need of using BigDecimal.
8. **Reports**: After discussing the Reports I assume this is acceptance criteria and specific reports:
   1. Daily incoming/outgoing report: Create one report of settlement dates with incoming and outgoing totals for each date sorted by ascending settlement date. Fields: Settlement date, incoming, outgoing.
   2. Ranking of entities based on incoming: Fields: entity name, incoming. Sorted by descending total incoming.
   3. Ranking of entities based on outgoing: Fields: entity name, outgoing. Sorted by descending total outgoing.

## Solution Design

This is just a summary of design decisions.

**Data structures**: The design is focused on reading the input once and counting the totals per entity and date in the meantime by using some data structure that are appropriate to keep a low time complexity.

**Design principles**: The design also encapsulates these data structures using well know patterns and following the SOLID principles. The design emerges from writing unit tests that fails, implementing and refactoring.

**YAGNI/Over engineering balance**: If we think of the “you aren’t gonna need it” principle, the design may look a little over engineered in some areas for the scope of the problem but it is more to show options of refactoring that would be applied in production code with more requirements.

**New requirements/flexibility**: The solution does not support requirements that are not needed now like working days only Monday and Wednesday. New requirements (or bugs) should be added starting with a unit test and implement or fix and refactor. The design should allow changes easily.

**Unit tests**: The unit tests cover the main logic of the application but trivial classes, getter and setter methods are not tested. Some tests are parameterized to test many combinations of data.

**Java doc/Comments**: I am not providing API or usage documentation in the code. All notes in the code are to describe programming decisions and alternatives.

**UML**: I prefer doing TDD and programming by intention and let the design emerge. I personally do not use UML to design. However, class diagrams are easy to generate so, the main package has some class diagram that may help to visualize and to discuss.

## Feedback

I look forward to receive feedback or meet to discuss the solution.